文件编号：NT/SSD/OSSP-16021

版 本：V1.1

修订状态：修改

**Java编码规范**

2016-07-05发布 2016-07-06实施

版 本 历 史

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 版本 | 状态 | 作者 | 参与者 | 起止日期 | 描述 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

状态：新建，增加，修改，删除

目录

[1 前言 6](#_Toc459108889)

[1.1 声明 6](#_Toc459108890)

[1.2 适用范围 6](#_Toc459108891)

[1.3 为什么要有编码规范 6](#_Toc459108892)

[1.4 文档目的 6](#_Toc459108893)

[1.5 术语说明 7](#_Toc459108894)

[2 源文件基础 7](#_Toc459108895)

[2.1 文件名 7](#_Toc459108896)

[2.2 文件编码：UTF-8 7](#_Toc459108897)

[2.3 特殊字符 8](#_Toc459108898)

[3 源文件结构 8](#_Toc459108899)

[3.1 许可证或版权信息/开头注释 9](#_Toc459108900)

[3.2 package语句 9](#_Toc459108901)

[3.3 import语句 10](#_Toc459108902)

[3.4 类声明 11](#_Toc459108903)

[4 格式规范 12](#_Toc459108904)

[4.1 大括号 12](#_Toc459108905)

[4.2 块缩进：4个空格 13](#_Toc459108906)

[4.3 一行一个语句 13](#_Toc459108907)

[4.4 列限制：120 13](#_Toc459108908)

[4.5 行限制 13](#_Toc459108909)

[4.6 自动换行 13](#_Toc459108910)

[4.7 空白 14](#_Toc459108911)

[4.8 用小括号分组 15](#_Toc459108912)

[4.9 具体结构的格式说明 15](#_Toc459108913)

[4.10 自动格式化功能 18](#_Toc459108914)

[5 命名规范 19](#_Toc459108915)

[5.1 对所有标识符都通用的规则 19](#_Toc459108916)

[5.2 标识符类型的规则 19](#_Toc459108917)

[5.3 驼峰式命名法(CamelCase) 22](#_Toc459108918)

[6 注释规范 23](#_Toc459108919)

[6.1 实现注释 23](#_Toc459108920)

[6.2 文档注释Javadoc 25](#_Toc459108921)

[7 日志记录规范 26](#_Toc459108922)

[7.1 目的 26](#_Toc459108923)

[7.2 日志类型 27](#_Toc459108924)

[7.3 日志级别 27](#_Toc459108925)

[7.4 日志记录规范 28](#_Toc459108926)

[8 异常处理规范 29](#_Toc459108927)

[8.1 异常该怎么处理 29](#_Toc459108928)

[8.2 规范 29](#_Toc459108929)

[8.3 建议 29](#_Toc459108930)

[9 单元测试规范（试用） 30](#_Toc459108931)

[9.1 规范 30](#_Toc459108932)

[10 编程惯例 31](#_Toc459108933)

[10.1 @Overrid 31](#_Toc459108934)

[10.2 静态成员 31](#_Toc459108935)

[10.3 禁用finalize方法 31](#_Toc459108936)

[10.4 去除警告 31](#_Toc459108937)

[10.5 字符串比较 31](#_Toc459108938)

[10.6 国际化 32](#_Toc459108939)

[10.7 编码转换 32](#_Toc459108940)

[11 附录 32](#_Toc459108941)

[11.1 eclipse需要导入的java相关配置 32](#_Toc459108942)

[11.2 修改注释中作者名为自己中文名字方法 32](#_Toc459108943)

[11.3 java代码保存时自动格式化 33](#_Toc459108944)

# 前言

## 声明

本文档是研发工程师的Java编程风格规范的完整定义。当且仅当一个Java源文件符合此文档中的规则时，我们才认为它符合的Java编程风格。

## 适用范围

本文档的适用范围是所有类型项目，包括合同项目和产品研发。

## 为什么要有编码规范

编码规范对于开发人员来说是非常重要的，有以下几个原因：

* 一个软件的生命周期中，80%的花费在于维护
* 几乎没有任何一个软件，在其整个生命周期中，均由最初的开发人员来维护
* 编码规范可以改善软件的可读性，可以让程序员尽快而彻底地理解新的代码
* 如果你将源码作为产品发布，就需要确任它是否被很好的打包并且清晰无误，一如你已构建的其它任何产品

## 文档目的

### 使代码更加规范

通过使用规范进行重构，可以使现有的代码得到规范的整理，使新代码在规范中产生。虽然不同的模块可能是由不同的人编写，但应该看上去差异不大。这样在阅读他人的代码时，就如同阅读自己的代码一样，减少了阅读代码的困难。

### 使注释更加规范

注释更加规范，使程序代码的可读性提高。可以生成风格统一的doc文档，在生成项目文档时可以节省时间。

### 使项目新成员能够快速上手

如果项目新成员进入开发，则可以通过规范文档了解项目的规范要求和常用工具的使用，快速的进入到开发角色中。

### 使系统易于维护

使用规范完成的程序，可阅读性大大提高，代码之间的关系将更加明朗，公用代码和标签将维护的工作大大降低，程序不稳定性也会降低。

### 使系统的性能有所提高

在编程规范中的有一些对性能提高要求的规范，这将使系统的性能有所改善。

### 使系统更加稳定

在规范中我们规定了测试代码的位置和编写要求，增强系统中白盒测试的作用。另外在编程惯例中也会提到一些规避出错的办法。这样使我们的系统更加稳定

## 术语说明

在本文档中，除非另有说明：

* 术语“类”可表示一个普通类、枚举类、接口或是annotaiton类型

# 源文件基础

## 文件名

源文件以其最顶层的类名来命名，大小写敏感，文件扩展名为.java。

## 文件编码：UTF-8

源文件编码格式为UTF-8，无Bom头的UTF-8，请不要eclipse以外的其它文本编辑器编辑源文件。

## 特殊字符

### 空白字符

除了行结束符之外，空格是源文件中唯一允许出现的空白字符。这意味着：

* 所有其它字符串中的空白字符都要进行转义。
* 制表符不用于缩进。

### 特殊转义序列

对于具有特殊转义序列的任何字符(\b, \t, \n, \f, \r, \”, \’及\)，我们使用它的转义序列，而不是相应的八进制(比如\012)或Unicode(比如\u000a)转义。

### 非ASCII字符

对于剩余的非ASCII字符，是使用实际的Unicode字符(比如∞)，还是使用等价的Unicode转义符(比如\u221e)，取决于哪个能让代码更易于阅读和理解。

注：在使用Unicode转义符或是一些实际的Unicode字符时，建议做些注释给出解释，这有助于别人阅读和理解。

例如：

String unitAbbrev = "μs"; | 赞，即使没有注释也非常清晰

String unitAbbrev = "\u03bcs"; // "μs" | 允许，但没有理由要这样做

String unitAbbrev = "\u03bcs"; // Greek letter mu, "s" | 允许，但这样做显得笨拙还容易出错

String unitAbbrev = "\u03bcs"; | 很糟，读者根本看不出这是什么

# 源文件结构

一个源文件包含(按顺序地)：

* 许可证或版权信息/开头注释
* package语句
* import语句
* 一个顶级类(只有一个)

以上每个部分之间用一个空行隔开。

## 许可证或版权信息/开头注释

如果一个文件包含许可证或版权信息，那么它应当被放在文件最前面。

其格式如下所示：

/\*

\* 版权所有 ( c ) 同方威视技术股份有限公司2015。保留所有权利。

\*

\* 本系统是商用软件，未经授权不得擅自复制或传播本程序的部分或全部

\*

\* 项目： PROJECT\_NAME V1.0（项目名称+ 版本号）

\* 文件名： Test.java

\* 描述： 【说明文件的功能】

\* 作者名： 【作者姓名】（中文姓名）

\* 日期： 【创建这个文档的时间】（包含年月日）

\*

\* 修改历史：

\* 【时间】 【修改者】 【修改内容】

\* 2000/12/26 ###（修改者） \*\*\*\*\*\*\*（修改内容，指修改方法、增加功能等）；

\* 2000/11/20 ###（修改者） \*\*\*\*\*\*\*（修改内容，指修改方法、增加功能等）；

\*

\*/

## package语句

package语句不换行，列限制并不适用于package语句。(即package语句写在一行里)

## import语句

### import不要使用通配符

即，不要出现类似这样的import语句：import java.util.\*;

### import语句不要换行

import语句不换行，列限制并不适用于import语句。(每个import语句独立成行)

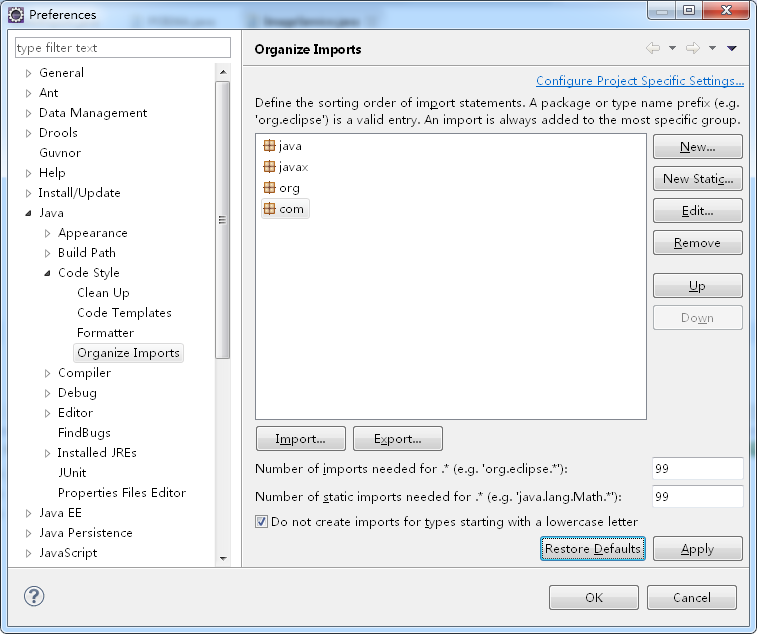
### 顺序和间距（不强制要求）

import语句可分为以下几组，按照这个顺序，每组由一个空行分隔：

* 所有的静态导入独立成组
* java imports
* javax imports
* 第三方的包。每个顶级包为一组，字典序。例如：android, com, junit, org, sun

组内不空行，按字典序排列。

其中包的引入顺序可在eclipse中做设置，参考界面如下：



## 类声明

### 只有一个顶级类声明

每个顶级类都在一个与它同名的源文件中。  
例外：package-info.java，该文件中可没有package-info类。

### 类成员顺序

类的成员顺序对易学性有很大的影响，但这也不存在唯一的通用法则。不同的类对成员的排序可能是不同的。

最重要的一点，每个类应该以某种逻辑去排序它的成员，维护者应该要能解释这种排序逻辑。比如， 新的方法不能总是习惯性地添加到类的结尾，因为这样就是按时间顺序而非某种逻辑来排序的。

建议顺序：

* 1. 静态变量
  + 公共变量
  + 保护变量
  + 无修饰符变量
  + 私有变量
* 2. 实例变量
  + 公共变量
  + 保护变量
  + 无修饰符变量
  + 私有变量
* 3. 构造方法
* 4. 方法
* 5. 内部类

#### 重载：永不分离

当一个类有多个构造函数，或是多个同名方法，这些函数/方法应该按顺序出现在一起，中间不要放进其它函数/方法。

# 格式规范

## 大括号

### 使用大括号

大括号与if, else, for, do, while等语句一起使用，即使只有一条语句或是空，也必须把大括号写上。

### 非空块：K & R 风格

对于非空块和块状结构，大括号遵循Kernighan和Ritchie风格 (Egyptian brackets)：

* 左大括号前不换行
* 左大括号后换行
* 右大括号前换行
* 如果右大括号是一个语句、函数体或类的终止，则右大括号后换行; 否则不换行。例如，如果右大括号后面是else、catch、finally或逗号，则不换行。

示例：

return new MyClass() {

@Override

public void method() {

if (condition()) {

try {

something();

} catch (ProblemException e) {

recover();

}

}

}

};

### 空块：可以用简洁版本

一个空的块状结构里什么也不包含，大括号可以简洁地写成{}，不需要换行。例外：如果它是一个多块语句的一部分(if/else 或 try/catch/finally) ，即使大括号内没内容，右大括号也要换行。

示例：

void doNothing() {}

## 块缩进：4个空格

每当开始一个新的块，缩进增加4个空格，当块结束时，缩进返回先前的缩进级别。缩进级别适用于代码和注释。(见4.1.2节中的代码示例)

## 一行一个语句

每个语句后要换行。

## 列限制：120

一个项目可以选择一行120个字符的列限制，除了下述例外，任何一行如果超过这个字符数限制，必须自动换行。

例外：

* 不可能满足列限制的行(例如，Javadoc中的一个长URL，或是一个长的JSNI方法参考)。
* package和import语句(见3.2节和3.3节)。
* 注释中那些可能被剪切并粘贴到shell中的命令行。

## 行限制

对java代码中类和方法的行数不做固定的限制，以易读、易理解为标准，不要在一个方法中写过长、过于复杂的逻辑，将复杂的逻辑进行必要的拆分。

## 自动换行

术语说明：一般情况下，一行长代码为了避免超出列限制(80或100个字符)而被分为多行，我们称之为自动换行(line-wrapping)。

我们并没有全面，确定性的准则来决定在每一种情况下如何自动换行。很多时候，对于同一段代码会有好几种有效的自动换行方式。

### 从哪里断开

自动换行的基本准则是：更倾向于在更高的语法级别处断开。

* 如果在非赋值运算符处断开，那么在该符号前断开，该符号位于下一行(比如在“+”处断开，“+”应该位于下一行)。这条规则也适用于以下“类运算符”符号：
  + 点分隔符“.”
  + 类型界限中的&（<T extends Foo & Bar>)
  + catch块中的管道符号(catch (FooException | BarException e)
* 如果在赋值运算符处断开，通常的做法是在该符号后断开(比如“=”应该与其前面的内容在同一行)。这条规则也适用于for循环语句中的分号。
* 方法名或构造函数名与左括号留在同一行。
* 逗号“,”与其前面的内容留在同一行。

### 自动换行时缩进4个空格

自动换行时，第一行后的每一行至少比第一行多缩进4个空格(注意：制表符不用于缩进。见2.3.1节)。

当存在连续自动换行时，缩进可能会多缩进不只4个空格(语法元素存在多级时)。一般而言，两个连续行使用相同的缩进当且仅当它们开始于同级语法元素。

## 空白

### 空行

以下情况需要使用一个空行：

* 类内连续的成员之间：字段，构造函数，方法，嵌套类，静态初始化块，实例初始化块。
  + 两个连续字段之间的空行是可选的，字段之间的空行主要用来对字段进行逻辑分组。
* 在函数体内，语句的逻辑分组间使用空行。
* 类内的第一个成员前或最后一个成员后的空行是可选的(既不鼓励也不反对这样做，视个人喜好而定)。
* 要满足本文档中其他节的空行要求(比如3.3节：import语句)

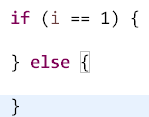
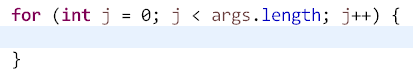
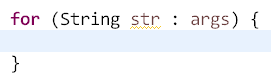
另外，尽量不要使用多个连续的空行。

### 单个空格

对单个空格在代码中出现的位置做以要求（不包含文字、注释和Javadoc中的空格）：

* 保留字和紧随其后的左括号“(”之间要有一个空格（如if, for catch等）。
* 右大括号“}”和其后面的保留字之间要有一个空格（如else, catch）。
* 在左大括号前要有一个空格，两个例外：
  + @SomeAnnotation({a, b})(不使用空格)。
  + String[][] x = {{“foo”}};(大括号间没有空格)。
* 在任何二元或三元运算符的两侧。这也适用于以下“类运算符”符号：
  + 类型界限中的：<T extends Foo & Bar>
  + catch块中的管道符号(catch (FooException | BarException e)。
  + for循环语句中的冒号。
* 在逗号、冒号、分号及右括号的后面
* 如果在一条语句后做注释，则双斜杠(//)两边都要一个空格。
* 类型和变量之间：List list。
* 数组初始化中，大括号内的空格是可选的，即new int[] {5, 6}和new int[] { 5, 6 }都是可以的。

示例：

## 用小括号分组

尽可能的在需要的地方加上小括号，除非去掉小括号能让代码更易于阅读并且不会被误解。我们没有理由假设读者能记住整个Java运算符优先级表。 如：最好把 a + b \* c + d 写成 a + (b \* c) + d

## 具体结构的格式说明

### 枚举类

枚举常量间用逗号隔开，换行可选。没有方法和注释的枚举类可写成数组初始化的格式：

private enum Suit { CLUBS, HEARTS, SPADES, DIAMONDS }

由于枚举类也是一个类，因此所有适用于其它类的格式规则也适用于枚举类。

### 变量声明

#### 每次只声明一个变量

不要使用组合声明，比如int a, b;。

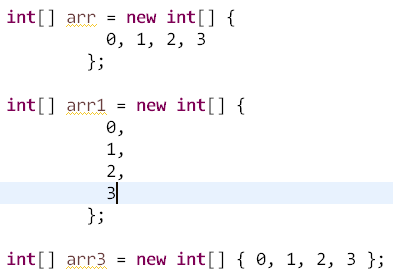
#### 需要时才声明，并尽快进行初始化

不要在一个代码块的开头把局部变量一次性都声明了(这是c语言的做法)，而是在第一次需要使用它时才声明。 局部变量在声明时最好就进行初始化，或者声明后尽快进行初始化。

### 数组

#### 数组初始化：可写成块状结构

数组初始化可以写成块状结构，比如，下面的写法都是OK的：



#### 数组声明

中括号是类型的一部分，而不是变量名的一部分：如String[] args， 而非String args[]。

### switch语句

switch块的大括号内是一个或多个语句组。

#### 缩进

与其它块状结构一致，switch块中的内容缩进为4个空格。每个switch标签后新起一行，再缩进4个空格，写下一条或多条语句。

#### 对于没有终止的case语句要加上注释说明

在一个switch块内，每个语句组要么通过break, continue, return或抛出异常来终止，要么通过一条注释来说明程序将继续执行到下一个语句组， 任何能表达这个意思的注释都是可以的(典型的是用// fall through)。这个特殊的注释并不需要在最后一个语句组(一般是default)中出现。

|  |
| --- |
| 示例 |
| **switch** (input) {  **case** 1:  **case** 2:  prepareOneOrTwo();  // fall through  **case** 3:  handleOneTwoOrThree();  **break**;  **default**:  handleLargeNumber(input);  } |

#### default语句不能省略

每个switch语句都包含一个default语句组，即使它什么代码也不包含。

### 注解(Annotations)

注解紧跟在文档块后面，应用于类、方法和构造函数，一个注解独占一行。这些换行不属于自动换行(第4.5节，自动换行)，因此缩进级别不变。

|  |
| --- |
| 示例 |
| @Override  @Nullable  public String getNameIfPresent() { ... } |

### 修饰符

类和成员的修饰符如果存在，则按Java语言规范中推荐的顺序出现。

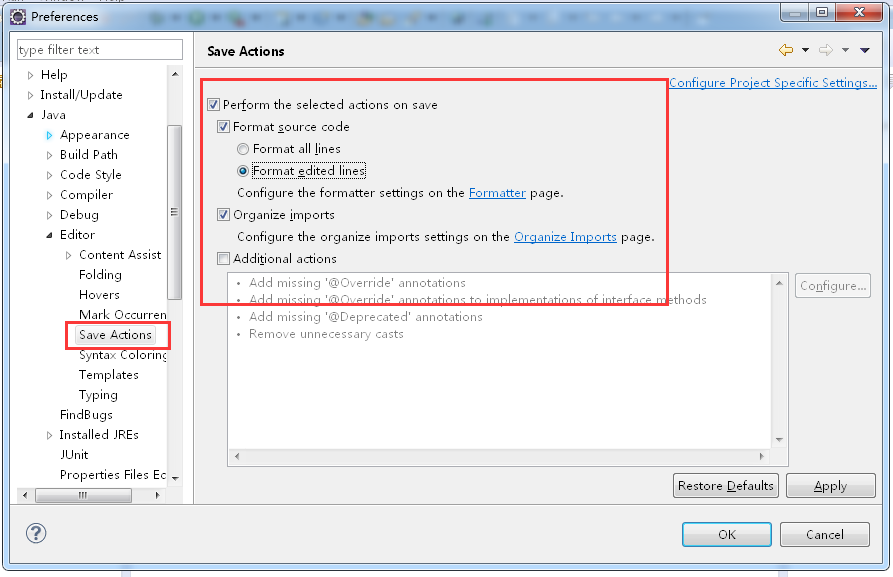
public protected private abstract static final transient volatile synchronized native strictfp

### 数值定义

长整型类型的数值用大写字母L结尾，不用小写字母（以避免与数字1混淆）。如：用1000000000L，而非1000000000l;

## 自动格式化功能

必须导入公共的配置，在导入公共的配置文件后方可使用eclipse中的ctrl+shift+F功能进行自动格式化代码。要求每个人提交的代码都必须满足格式化要求。可在eclipse中设置保存文件时对修改的代码进行自动格式化，如下图所示：



# 命名规范

## 对所有标识符都通用的规则

命名时必须用英文单词或常见的缩写，不能用汉语拼音进行命名。若不知道一个词的英文怎么写，可以先通过翻译软件进行翻译，然后记住它的写法。

标识符只能使用字母和数字，因此每个有效的标识符名称都能匹配正则表达式“\w+”。

## 标识符类型的规则

### 包名

包名全部小写，连续的单词只是简单地连接起来，不使用下划线。

一般规则：com.nuctech.<项目名>.<模块名>.<action/service/dao/model>

### 类名

类名都以大驼峰式命名法（UpperCamelCase）风格命名。

类名通常是名词或名词短语，接口名称有时可能是形容词或形容词短语。

#### 测试类

测试类的命名以它要测试的类的名称开始，以Test结束。例如，HashTest或HashIntegrationTest。

#### 接口类

当要区别接口和实现类的时候，在实现类的后面加上“Impl”。例如：

interface Container

class ContainerImpl

#### 异常类

Exception类最好能用“Exception”做为类命名的结尾。例如：

DataNotFoundException

InvalidArgumentException

#### 抽象类

抽象类最好能用“Abstract”做为类命名的开头。例如：

AbstractBeanDefinition

AbstractBeanFactory

#### 工厂类

工厂类最好能以“Factory”作为类命名的结尾，工厂方法最好能把该方法做要创建的对象类型描述出来。例如：

public ContainerFactory createContainer();

### 方法名

方法名都以小驼峰式命名法（lowerCamelCase）风格命名。

方法名通常是动词或动词短语。

### 常量名

常量名全部为大写字母，每个单词间用下划线分隔开。

每个常量都是一个静态final字段，但不是所有静态final字段都是常量。在决定一个字段是否是一个常量时， 考虑它是否真的是一个常量。只是永远不打算改变的对象一般是不够的，它要真的一直不变才能将它示为常量。

// Constants

static final int NUMBER = 5;

static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");

static final Joiner COMMA\_JOINER = Joiner.on(','); // because Joiner is immutable

static final SomeMutableType[] EMPTY\_ARRAY = {};

enum SomeEnum { ENUM\_CONSTANT }

// Not constants

static String nonFinal = "non-final";

final String nonStatic = "non-static";

static final Set<String> mutableCollection = new HashSet<String>();

static final ImmutableSet<SomeMutableType> mutableElements = ImmutableSet.of(mutable);

static final Logger logger = Logger.getLogger(MyClass.getName());

static final String[] nonEmptyArray = {"these", "can", "change"};

### 非常量字段名

非常量字段名以小驼峰式命名法（lowerCamelCase）风格命名。这些名字通常是名词或名词短语。

### 参数名

参数名以小驼峰式命名法（lowerCamelCase）风格命名。参数应该避免用单个字符命名。

### 局部变量名

局部变量名以小驼峰式命名法（lowerCamelCase）风格命名，比起其它类型的名称，局部变量名可以有更为宽松的缩写。

虽然缩写更宽松，但还是不要用单字符进行命名，除了临时变量和循环变量。

即使局部变量是final的不可改变的，也不应该把它示为常量，自然也不能用常量的规则去命名它。

### 泛型变量名

泛型变量可用以下两种风格之一进行命名：

* 单个的大写字母，后面可以跟一个数字(如：E, T, X, T2)。
* 以类命名方式(5.2.2节)，后面加个大写的T(如：RequestT, FooBarT)。

## 驼峰式命名法(CamelCase)

驼峰式命名法分大驼峰式命名法(UpperCamelCase)和小驼峰式命名法(lowerCamelCase)。 有时，我们有不只一种合理的方式将一个英语词组转换成驼峰形式，如缩略语或不寻常的结构(例如"IPv6"或"iOS")。我们要按以下要求进行转换：

* 把短语转换为纯ASCII码，并且移除任何单引号。例如："Muller’s algorithm"将变成"Muellers algorithm"。
* 把这个结果切分成单词，在空格或其它标点符号(通常是连字符)处分割开。
  + 推荐：如果某个单词已经有了常用的驼峰表示形式，按它的组成将它分割开(如"AdWords"将分割成"ad words")。 需要注意的是"iOS"并不是一个真正的驼峰表示形式，因此该推荐对它并不适用。
* 现在将所有字母都小写(包括缩写)，然后将单词的第一个字母大写：
  + 每个单词的第一个字母都大写，来得到大驼峰式命名。
  + 除了第一个单词，每个单词的第一个字母都大写，来得到小驼峰式命名。
* 最后将所有的单词连接起来得到一个标识符。

示例：

|  |  |  |
| --- | --- | --- |
| **Prose form** | **Correct** | **Incorrect** |
| "XML HTTP request" | XmlHttpRequest | XMLHTTPRequest |
| "new customer ID" | newCustomerId | newCustomerID |
| "inner stopwatch" | innerStopwatch | innerStopWatch |
| "supports IPv6 on iOS?" | supportsIpv6OnIos | supportsIPv6OnIOS |
| "YouTube importer" | YouTubeImporter YoutubeImporter\* |  |

加星号处表示可以，但不推荐。

注：在英语中，某些带有连字符的单词形式不唯一。例如："nonempty"和"non-empty"都是正确的，因此方法名checkNonempty和checkNonEmpty也都是正确的。

# 注释规范

Java程序有两类注释：实现注释（implementation comments）和文档注释（document comments）。实现注释是使用“/\*……\*/”和“//”界定的注释。文档注释是使用“/\*\*……\*/”界定的，并可以通过javadoc工具转换成HTML文件。

注释应该被用来给出代码的总括，并提供代码自身没有提供的附加信息。注释应该仅包含与阅读和理解程序有关的信息。

注释不应该写在星号或其它字符画出来的大框里。

Java代码中建议由20%的注释，20%行的空行，便于阅读。

使用中文进行注释，注释中的作者名必须使用中文名字，除非你没有中文名字。

## 实现注释

### 文件注释

其格式如下所示：

/\*

\* 版权所有 ( c ) 同方威视技术股份有限公司2015。保留所有权利。

\*

\* 本系统是商用软件，未经授权不得擅自复制或传播本程序的部分或全部

\*

\* 项目： PROJECT\_NAME V1.0（项目名称+ 版本号）

\* 文件名： Test.java

\* 描述： 【说明文件的功能】

\* 作者名： 【作者姓名】（中文姓名）

\* 日期： 【创建这个文档的时间】（包含年月日）

\*

\* 修改历史：

\* 【时间】 【修改者】 【修改内容】

\* 2000/12/26 ###（修改者） \*\*\*\*\*\*\*（修改内容，指修改方法、增加功能等）；

\* 2000/11/20 ###（修改者） \*\*\*\*\*\*\*（修改内容，指修改方法、增加功能等）；

\*

\*/

### 块注释风格

块注释与其周围的代码在同一缩进级别。它们可以是/\* ... \*/风格，也可以是// ...风格。对于多行的/\* ... \*/注释，后续行必须从\*开始， 并且与前一行的\*对齐。以下示例注释都是OK的。

/\*

\* This is // And so /\* Or you can

\* okay. // is this. \* even do this. \*/

\*/

注释不要封闭在由星号或其它字符绘制的框架里。

注：在写多行注释时，使用/\* ... \*/。

### 其它注释规范

以下不写入Java Doc，用于帮助阅读代码，采用“//”注释：

* 必须有步骤（流程）注释：详设中有关流程的说明可作为注释加入适当的位置；
* 必须有算法说明：有关算法的简要说明；
* 重要的循环、分支的判别条件必须有说明；
* 尚未完成或有问题的代码段，必须在后面加上如下注释：//TODO
* 不许在注释中使用不常用的缩写；
* 注释必须与其描述的代码相近，对代码的注释应放在其上方或右方相邻位置，不可放在下面，如放于上方则需与其上面的代码用空行隔开；如放于右方则表示此注释仅针对本条语句。
* 对变量的注释应放在其上方相邻位置，不可放在下面；对类中的每个字段的注释可放在此字段的右方；
* 在类说明中，如果涉及类中各方法协调工作，还要说明怎样协调.
* 如果必要，注明注意事项，和关联关系
* 如果有人在修改其他人创建的代码时，也应该添加必要的注释。必要时需要在注释中说明修改代码的时间、人员，具体情况等

## 文档注释Javadoc

### 格式

#### 一般形式

Javadoc的基本格式如下所示：

/\*\*

　\* 注释内容，说明这个类的功能，用途，内容等

　\*

　\* @author ####

\*/

public int method(String p1) { ... }

或者是以下单行形式：

/\*\* An especially short bit of Javadoc. \*/

基本格式总是OK的。当整个Javadoc块能容纳于一行时(且没有Javadoc标记@XXX)，可以使用单行形式。

#### 段落

空注释行会出现在段落之间和Javadoc注释标记(@XXX)之前。 除了第一个段落，每个段落第一个字前都有标签<p>，并且它和第一个字间没有空格。

#### Javadoc标记

标准的Javadoc标记按以下顺序出现：@param, @return, @throws, @deprecated, 前面这4种标记如果出现，描述都不能为空。 当描述无法在一行中容纳，连续行需要至少再缩进4个空格。

如果参数或返回值是特定的格式，需要在描述中加以详细的说明。例如：方法的返回值是一个xml结构，则需要在描述中说明此xml结构的定义或参看的定义文档。

### 简要描述

每个类或成员的Javadoc都以一个简要的描述或说明开始。这个描述是非常重要的，在某些情况下，它是唯一出现的文本，比如在类和方法索引中。

这只是一个简要的描述，可以是一个名词短语或动词短语，但不一定是一个完整的句子。

注：一个常见的错误是把简单的Javadoc写成/\*\* @return the customer ID \*/，这是不正确的。它应该写成/\*\* Returns the customer ID. \*/。

### 哪里需要使用Javadoc

至少在每个public类及它的每个public和protected成员处使用Javadoc，以下是一些例外：

#### 例外：不言自明的方法

对于简单明显的方法如setter、getter方法，Javadoc是可选的。

单元测试类中的测试方法中的Javadoc也是可选的，我们通常可以从这些方法的命名中知道它是干什么的，因此不需要额外的文档说明。

注：如果有一些相关信息是需要后续的开发人员了解的，那么以上的例外不应作为忽视这些信息的理由。

#### 例外：重载

如果一个方法重载了超类中的方法，那么Javadoc并非必需的。

#### 可选的Javadoc

对于包外不可见的类和方法，根据需要添加 Javadoc。

# 日志记录规范

## 目的

定义日志记录规范的目的：

* 记录系统运行异常信息
* 记录系统运行状态信息
* 记录系统运行性能信息
* 记录系统运行调试信息
* 日志记录的真正作用在于当有问题发生时，能够帮助开发人员很快的定位问题所在。

## 日志类型

日志主要分为三大类：

* 安全类信息：记录系统边界交互行为和信息
* 业务类信息：记录系统内容业务处理行为和信息
* 性能类信息：记录系统硬件对业务处理的支撑能力

## 日志级别

日志信息输出的优先级从高到低至少应分为五档，分别是FATAL、ERROR、WARN、INFO、DEBUG，以 ERROR、WARN、INFO 和 DEBUG 作为常用。这些级别用来指定这条日志信息的重要程度。在测试阶段可以打开所有级别的日志，系统上线后只允许输出INFO以上级别（含INFO）。各级别的日志信息作用规定如下：

* 致命（Fatal）

 严重的错误，系统无法正常运行，如硬盘空间满等。这个级别很少被用，常暗含系统或者系统的组件迫近崩溃。

* 错误（Error）

系统可以继续运行，但最好要尽快修复的错误。这个级别用的较多，常常伴随Java运行时异常及预期之外的错误，但系统可以继续运行。

* 警告（Warn）

系统可以正常运行，但需要引起注意的警告信息。这个级别预示较小的问题，由系统外部的因素造成的，比如用户输入了不符合条件的参数。

* 信息（Info）

 系统运行的主要关键时点的操作信息，一般用于记录业务日志。但同时，也应该有足够的信息以保证可以记录再现缺陷的路径。这个级别记录了系统日常运转中有意义的事件。

* 调试（Debug）

系统运行中的调试信息，便于开发人员进行错误分析和修正，一般用于程序日志，关心程序操作(细粒度)，不太关心业务操作(粗粒度)。

## 日志记录规范

### 框架

用log4j做为日志记录的框架；在项目的源文件目录下对日志做相关配置。

### 规范

#### 必须

1. 一个对象中通常只使用一个Logger，Logger应该是static final的。定义如下：

private static final Logger logger = Logger.getLogger(ClassName.class);

1. 输出Exceptions的全部Throwable信息，不能丢失StackTrace信息

logger.error("some info",e);

1. 系统出现问题时，必须抛出异常，在处理异常时记录日志，且日志级别必须是前三个级别（Fatal\Error\Warning）中的一种。
2. 有意义的日志，通常情况下在程序日志里记录一些比较有意义的状态数据：程序启动，退出的时间点；程序运行消耗时间；耗时程序的执行进度；重要变量的状态变化。
3. 日志输出中的变量应当加以判断，不要引起NullPointerException等，避免在日志记录语句中出现错误。
4. 日志中包含充分的信息。在记录日志消息时应该尽可能多的包含当前上下文中的各种信息，以方便在遇到问题时可以快速的获取到所需的信息。比如在网上支付功能中，与支付相关的日志应该完整的包含当前用户、订单以及支付方式等全部信息。一种比较常见的问题是把相关的日志记录分散在由不同日志记录器所记录的日志中。当出现问题之后，需要手工查找并匹配相关的日志来定位问题，所花费的时间和精力会更多。因此，应该尽可能在一条日志记录中包含足够多的信息。
5. 对于不重要的日志信息，请调低日志信息的级别。

#### 禁止用以下方式记录日志

1. 不允许出现System print(包括System.out.println和System.err.println)语句。
2. 不允许出现printStackTrace。
3. 对于“敏感”的信息，如客户密码等信息，禁止使用debug以上级别的日志记录，在开发过程中，也应减少日志对该类信息的记录。

#### 建议

日志记录时，尽量不要在参数中拼日志信息，可用StringBuffer或String.format()方法将日志信息拼好后再传入日志记录方法中。

# 异常处理规范

## 异常该怎么处理

处理异常主要有四个选择：

　　1、处理异常。针对该异常采取一些行动，例如修正问题、提醒某个人或进行其他一些处理，要根据具体的情形确定应该采取的动作。仅记录日志算不上已经“处理好了异常”。

　　2、重新抛出异常。处理异常的代码在分析异常之后，认为自己不能处理它，重新抛出异常也不失为一种选择。

　　3、把该异常转换成另一种异常。大多数情况下，这是指把一个低级的异常转换成应用级的异常（其含义更容易被用户了解的异常）。

　　4、不要捕获异常。

既然捕获了异常，就要对它进行适当的处理。不要捕获异常之后又把它丢弃，不予理睬。

## 规范

1. 当程序产生异常时，必须捕捉并处理异常、并将异常记录到日志中（除非打算抛出异常）。
2. 在异常处理模块中提供适量的错误原因信息。
3. 使用finally避免资源泄露。保证所有资源都被正确释放，充分运用finally关键词。
4. 不要从try区段中返回。
5. 不要在 finally 内部使用 return 语句。它不仅会影响函数的正确返回值，而且它可能还会导致一些异常处理过程的意外终止，最终导致某些异常的丢失。
6. 将try/catch区段置于循环之外。
7. 不要将异常用于程序流程控制。
8. 一个方法不应抛出太多类型的异常。说明：如果程序中需要分类处理，则将异常根据分类组织成继承关系。如果确实有很多异常类型首先考虑用异常描述来区别，throws/exception子句标明的异常最好不要超过三个。
9. 不要定义 Error 和 RuntimeException 的子类，可以定义 Exception 的子类。
10. 在catch语句中尽可能指定具体的异常类型，必要时使用多个catch。不要试图处理所有可能出现的异常，即不要用Exception去捕获异常。

## 建议

1. 在捕获异常的时候，不使用 Exception, RuntimeException, Throwable，尽可能使用它们的子类。

2. 异常捕获尽量不要直接 catch (Exception ex)，应该把异常细分处理。

3. 在程序中使用异常处理还是使用错误返回码处理，根据是否有利于程序结构来确定，并且异常和错误码不应该混合使用，推荐使用异常。

4. 尽量减小try块的体积。

经常可以看到有人把大量的代码放入单个try块，实际上这不是好习惯。这种现象之所以常见，原因就在于有些人图省事，不愿花时间分析一大块代码中哪几行代码会抛出异常、异常的具体类型是什么。把大量的语句装入单个巨大的try块就象是出门旅游时把所有日常用品塞入一个大箱子，虽然东西是带上了，但要找出来可不容易。

一些新手常常把大量的代码放入单个try块，然后再在catch语句中声明Exception，而不是分离各个可能出现异常的段落并分别捕获其异常。这种做法为分析程序抛出异常的原因带来了困难，因为一大段代码中有太多的地方可能抛出Exception。

5. 尽量抛出异常，顶层的方法截获所有的异常，并且记录在日志中，根据需要也可反馈给上层应用。

# 单元测试规范（试用）

我们在编写大型程序的时候，需要写成千上万个方法或函数，这些函数的功能可能很强大，但我们在程序中只用到该函数的一小部分功能，并且经过调试可以确定，这一小部分功能是正确的。但是，我们同时应该确保每一个函数都完全正确，因为如果我们今后如果对程序进行扩展，用到了某个函数的其他功能，而这个功能有bug的话，那绝对是一件非常郁闷的事情。所以说，每编写完一个函数之后，都应该对这个函数的方方面面进行测试，这样的测试我们称之为单元测试。

程序员要对自己编写的代码负责，您不仅要保证它能通过编译，正常地运行，而且要满足需求和设计预期的效果。单元测试正是验证代码行为是否满足预期的有效手段之一。

## 规范

此规范目前处于试用阶段，不做强制要求。

* 使用Junit进行单元测试
* 不要使用main方法进行测试，必须创建专门的测试类
* 在项目中创建与src同级的test目录，将测试代码全部放入其中
* test目录下的包结构应与src下的包结构命名一致
* 测试类的命名规范参看5.2.2.1
* 建议对核心的业务方法做单元测试
* 通用组件必须做单元测试

# 编程惯例

## @Overrid

所有重写的方法都必须加上这个标注，添加这个标注的目的：

* 帮助自己检查是否正确的复写了父类中已有的方法
* 告诉读代码的人，这是一个复写的方法

## 静态成员

必须使用类名来调用静态的类成员，而不是用具体的某个对象或表达式

## 禁用finalize方法

在任何时候都不要调用此方法。不使用此方法的原因：

* Finalizers是不可预知的，有时是危险的，并且是不必要的。
* 使用finalize会导致严重的内存消耗和性能损失
* JVM不保证finalize一定会被执行，而且执行finalize的时间也不确定。
* 尽量不要使用System.gc或者System.runFinalization，它们导致finalizers的运行怪异，并且不提供任何保障。

## 去除警告

在eclipse中警告会以感叹号标出，请尽量去除。

警告可能有以下几种类型：

* + 导入未使用的类，请删除
  + 未使用的变量，请删除
  + 实现序列化的类，没有版本信息，请添加默认版本信息
  + 没有使用泛型，请加上泛型。例如：List<String> list = new ArrayList<String>()

## 字符串比较

字符串比较不能用“= = 或!=”操作符，要用equals方法

正确用法如下：“something”.equals(s)

错误用法：s.equsls(“something”) ，可能会有空指针异常

## 国际化

在向页面传递的字符串的Java代码中不允许直接出现中文字符。否则在做国际化产品时，会造成混乱并且添加了不必要的工作量。

## 编码转换

不允许在程序中有硬编码转换。例如下面的形式是不允许出现的。

condition = new String(condition.getBytes("ISO-8859-1"),"UTF-8");

# 附录

## eclipse需要导入的java相关配置

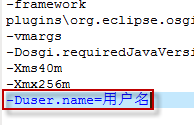
其中文件名，即为在eclipse选项中的菜单项路径,具体文件也可以访问 <http://172.16.254.102/svn/Svn_Repository/PublicResource/流程培训资料/单元测试和代码走查培训/checkstyle规范性检查/代码模板配置> 。

* java-codestyle-cleanup
* java-codestyle-codetemplates
* java-codestyle-formatter
* java-codestyle-organizeimports
* javascript-codestyle-cleanup
* javascript-codestyle-codetemplates
* javascript-codestyle-formatter

## 修改注释中作者名为自己中文名字方法

在eclipse/myeclipse中，当我们去添加注释的作者选项时，@author 后边一般都会默认填充的你登录计算机的用户名。如何去修改呢，建议通过修改ini配置文件来修改成自己的中文名字。

在eclipse/myeclipse的目录下找到eclipse.ini/myeclipse.ini文件，在-vmargs后边添加上启动参数：-Duser.name=你想要显示的名字。如下图：



## java代码保存时自动格式化

勾选以下选项，可以在保存java文件时，自动格式化代码

